New MIDAD Design

Brett Viren

bv@bnl.gov

Brookhaven National Lab

Talk Outline

1. Strategy of new MIDAD framework.
2. MVC implementation
3. Gui wrappers
   (a) libsigc++ signal/slots
   (b) libsigc++ memory management
4. Scenes, Scenery, SceneElements
   (a) How to.
5. Displays
6. Ranges
7. NamedFactory and NamedProxy
8. Still to do
9. MIDAD Demo

Strategy of new MIDAD Framework

- Continue to follow Model-View-Control (MVC) pattern.

Strategy of new MIDAD Framework

- Continue to follow Model-View-Control (MVC) pattern.
- Use ROOT for graphics and GUI, but otherwise keep it at bay. Only use where necessary, wrap and sanitize where possible. Minimize the need to run `rootcint`.

Strategy of new MIDAD Framework

- Continue to follow Model-View-Control (MVC) pattern.
- Use ROOT for graphics and GUI, but otherwise keep it at bay. Only use where necessary, wrap and sanitize where possible. Minimize the need to run `rootcint`.
- Rely on libsigc++ for signal/slots and memory management.

Strategy of new MIDAD Framework

- Continue to follow Model-View-Control (MVC) pattern.
- Use ROOT for graphics and GUI, but otherwise keep it at bay. Only use where necessary, wrap and sanitize where possible. Minimize the need to run `rootcint`.
- Rely on libsigc++ for signal/slots and memory management.
- Most classes are in `namespace Midad`. Exception for `Gui` related, including `Range` (separate lib one day?) and JobControl interface (`JOBCMODULE` CPP macro didn't like namespaces).

Model-View-Control (MVC)

Model

- Encapsulates a type of data (*eg.* a `CandHandle`) plus meta data.
- Exports an API to modify that data.
- Provides `modified` signal.
- `class CandModel<CandHandleType>`
  base class manages handle, calls `bool Update()` when new MomNavigator is set, if `true` returned, emits modified().
- One concrete example:

```
class DigitListModel
      : public CandModel<CandDigitListHandle>
```

View

- Implements a representation of a Model.
- `View` base class is templated on Model type.
- Attaches to Models `modified` signal.
- Calls `virtual Configure(ModelType&)` when Model is modified.
- Manages Model (via `SigC::Ptr`).
- Concrete View implements `Configure` to reconfigure itself.
- Concrete View typically subclass some other graphical class

Control

- Modifies a Model.
- `Control` base class is templated on Model type.
- Manages Model (via `SigC::Ptr`)
- Concrete class is typically owned by some other (possibly graphical) class.
- A class implementing a View can still have Controls.
- Should trigger `global_update` signal at the end of the modification of the Model (allows compound commands, This needs work).

Gui and libsigc++

Gui wrappers

The `Gui*` classes wrap ROOT's `TG*` classes in order to:

- Clean up and simplify the TG interface
- Provide libsigc++ signals
- Provide child widget memory management (via `SigC::Ptr`)

Features:

- Gtkmm design style loosely followed
- Children are passed by **reference** to parents, lets Widgets to be created on stack or heap
- TGLayoutHints are mostly unneeded or simplified
- Menu creation is much simpler
- Sliders, etc, use Ranges

libsigc++ signal/slots

Any useful Rt signals from the TG interface are exported as libsigc++ signals. Rt signals continue to work, but not used by MIDAD.

- In the Rt method, Signal must be a TQObject, slot must be a ROOTified object:

```
tqobj.Connect("ItsSig()","SomeClass",&rootobj,"AMethod()");
```

- With sigc++, any object can hold a signal object and the slot can be either a SigC::Object or a generic object:

```
any_obj.its_sig.connect(slot(an_obj,&SomeClass::AMethod));
any_obj.its_sig.connect(slot_class(an_obj,&SomeClass::AMethod));
```

Benifits of libsigc++

Use the magic of `bind`:

```
SigC::Signal0<void> SomeObject::its_signal;
void Class::Method(int);
any_obj.its_sig.connect(bind(slot(*ptr,&Class::Method),42));
```

Slots are first class objects:

```
SigC::Slot0<void> make_slot(void);
any_obj.its_sig.connect(make_slot());
```

Chain slots together:

```
void set(int i) { g_something = i; }
int get() { return g_somethingelse; }
SigC::Signal0<void> sig;
sig.connect(SigC::chain(SigC::slot(set),SigC::slot(get)));
```

Benifits of libsigc++ cont.

Save the connection:

```
Connection con = obj->its_signal.connect(slot(some_func));
obj->its_signal.emit(); // some_func() is called
con.block(true);
obj->its_signal.emit(); // some_func() is not called
con.block(flase);
obj->its_signal.emit(); // some_func() is called
con.disconnect();
obj->its_signal.emit(); // some_func() is not called
```

Chain signals:

```
Signal0<void> sig1, sig2;
sig1.connect(slot(some_func));
sig2.connect(sig1.slot());
sig2.emit(); // some_func() is called via sig2
             // triggering sig1’s emittance
```

libsigc++ memory management

Templated reference counted smart pointer (`SigC::Ptr<>`) used internally to libsigc++, but exported. When `Ptr` is deleted, so shall object if it was managed and no other `Ptrs` reference it:

```
{
  SigC::Ptr<SomeObject> obj;
  {
    SigC::Ptr<SomeObject> obj1 = manage(new SomeObject());
    SigC::Ptr<SomeObject> obj2 = manage(new SomeObject());
    SomeObject* obj3 = manage(new SomeObject());
    obj = obj2
    // obj1, obj2 lives
  }
// obj1 is deleted, obj2 lives on, obj3 is leaked
SomeObject obj4;
obj = &obj4;  // ok.  obj1 is now deleted.
obj = manage(new SomeObject()); // ok. obj4 is not deleted
}
```

Scenes, Scenery, SceneElements

Scene A TPad which knows about and manages Scenery. Contains a Scene type. Calls `TPad::Modified` in response to Scenery's modified signal. Calls `TPad::Update` in response to `global_update`.

Scenery Intended as the TPad/ROOT side of a graphical View. It is a TObject which draws into a Scene, possibly via other TObjects or SceneElements. Most concrete Scenery will also subclass some View. Emitts modified signal.

SceneElements Additional API for objects going into a Scene. Allows for, `eg.` pointer interaction. Most graphical "primitive" objects will inherit from this as well as some complex graphical TObject, eg TBox.

Scenery HOWTO

1. Inherit Scenery and (probably) some `View<ModelType>`
2. In `.cxx` file:

```
static Midad::SceneryProxy<Midad::YourConcreteScenery>
  gsYourConcreteSceneryProxy("Scenery::YourConcrete");
```

3. Implement `Configure(ModelType& mymodel)`

```
this->ClearPrimitives();
// Query Scene:
PlaneView::PlaneView_t pv = this->GetScene()->GetViewType();
// Add TObject or TObject + SceneElement
this->AddPrimitives(new TBox(0.2,0.2,0.8,0.8));
this->AddPrimitives(new MyConcreteSceneElement(mymodel));
// ...
```

Displays

- The Display base provides a widget with a menu, button and status bar as well as a central widget in which subclasses place things.
- Subclasses are aggregations of Scenes, Gui widgets or Gui Views.
- Currently one Display implemented: SceneDisplay holding a single Scene and a couple of GuiSliders.

Ranges

- Simple templated encapsulations of a min and max value.
- Follows MVC pattern (sort of).
- Emits `modified` signal when min/max change.
- Template gives type of min/max value. RangeDouble typedef most used.
- Usually shared via `SigC::Ptr`
- Used by Scenes for X-Y zooming (eventually color scale), Scenery for bounds, `GuiSlider`, Models for state data.

NamedFactory and NamedProxy

- NamedFactory is a singleton map of strings to NamedProxy’s.
- Concreate NamedProxy’s register themselves at link time.
- Proxies are looked up at run time to list (menu) and create subsystem objects.
- Concrete proxies have a ConcreteProxyBase class and a templated ConcreateProxy
- Concrete proxies can hold lists of all instantiated objects and emit signals when more objects are created or more proxies are linked in.
- Currently Displays and Scenery use proxies.

To do - near term

- Currently Concrete Proxies use a Create() method, want to move to a more general SigC::Slot based scheme.
- Implement more scenery (DigitList and TrackList so far, but lacking some features).
- Work out configuration scheme - all interactive now, friendly but tedious.
  - Export enough MIDAD API and do it all in a ROOT .C?
  - Use DBI/Registry mechanism?
  - Some custom config language? XML?
- Implement any missing functionality in the first MIDAD
- Work through the remaining “to do”s from the first MIDAD.

To do - far term

- Work out how to “connect” different data, (*eg.* Click on a hit, all other hits in track/shower light up
- MC objects, wait for Hugh.
- Job path MVC, Job module config interface.

MIDAD Demo

Cross your fingers.